Journal of Computational and Applied Mathematics 31 (1990) 153–163
North-Holland

Thoughts on pseudorandom number generators

B.D. RIPLEY
Department of Statistics, University of Strathclyde, 26 Richmond Street, Glasgow, UK G1 1XH

Received 15 January 1989

Abstract: Much of the informal discussion at the Workshop concerned the merits of different pseudorandom number generators. Here we record some comments based on comparing generators across a wide range of machines.

Keywords: Congruential generator, discrepancy, lattice test, shift-register generator.

1. What is a good generator?

The ideal properties of a good general-purpose pseudorandom number generator are easy to agree but impossible to achieve simultaneously. They include

- a very good approximation to a uniform distribution,
- very close to independent output in a moderate number of dimensions,
- repeatability from a simply specified starting point,
- speed,
- a very long period (at least 2^{50}).

The first two of these are axiomatic (or a definition of *pseudorandom*) since pseudorandom numbers will be used as if they were a realization of an independent $U(0, 1)$ stream. Repeatability is very important for debugging (in particular, debugging the work of others) and for understanding a stochastic process. For example, Ripley and Kirkland [14] show some summaries of the simulation of a Markov random field which show an abrupt change at one point in the supposedly converging iterative process. Because a repeatable sequence was used, the process could be run up to just before that point and stopped, so the critical phase could be examined in detail.

The last two points are also closely linked. I believe the time taken for pseudorandom number generation should be negligible compared to the operations performed on the numbers produced. Some of our iterative simulations on a Sun workstation spend 30% of the time generating numbers from a rather fast (13 μsec /call) generator, and so use 1.5 million random numbers per minute. A single run can easily take 10 hours or 9×10^8 random numbers. The period of many commonly used generators is around 2^{31}, one quarter of this number. Simulations in statistical physics on supercomputers can use very much larger numbers. There need be no great disadvantage in repeating the sequence within a simulation provided that it can be guaranteed to

be done in an asynchronous way (such as when the simulation is nonstationary). The real need for a long period is to satisfy the first two requirements simultaneously, since a relatively small number N of k-tuples cannot fill $[0, 1]^k$ adequately. An argument based on spatial statistics [13, p.26] suggests that we need $N \gg 200n^2$ if there are n iid points in the problem. Thus for small problems with up to 1000 points, a period of about 2^{31} is adequate, but for problems with a million or more points, periods of 2^{50} or more are necessary.

Even in more mundane applications a fast generator can avoid having to construct complicated nonuniform variate generators, either to minimize the number of uniforms used or by "reusing" uniforms. On modern hardware with standard functions built into maths coprocessor chips the elementary functions can be fast, so simple routines can also be very fast. This is illustrated in Table 2, which shows on the Sun 3 chipsets the calculation of a logarithm is faster than two calls to any pseudorandom number generator.

One solution to the problems of finding a good pseudorandom number generator is to use a machine-readable record of bytes produced by a physical device, as suggested by Luc Devroye. A laser disc could store of the order of 10^9 such bytes and can be addressed randomly, so a suitable "seed" would be the starting address. Although this may provide fewer random real numbers than we need, the sequence could be reused in some "random" way. The problems are at one extreme to deliver the numbers fast enough in certain sorts of parallel computers, and at the other, cost of the drive and intermediate storage. Physical devices are also used; the Institute of Statistical Mathematics in Tokyo has a 200 kbytes/sec physical generator, but this is of course not repeatable.

Park and Miller [10] comment that examples of good generators are hard to find, and suggest *LGM* (below) as a "minimal standard". Their search was, however, in the computer science literature, and mainly in texts at that; random number generation seems to be one of the most misunderstood subjects in computer science!

2. Some example generators

Congruential generators

Most commonly used generators are from the *linear congruential* family,

$$X_i = (aX_{i-1} + c) \bmod M, \qquad U_i = X_i/M, \tag{1}$$

with widely used examples including those mentioned in Table 1.

Table 1

Name	a	c	M	Period	Reference
LGM	16807	0	$2^{31}-1$	$2^{31}-2$	Lewis, Goodman and Miller [7]
PRB	630360016	0	$2^{31}-1$	$2^{31}-2$	Payne, Rabung and Bogyo [11]
Marsg	69069	1	2^{32}	2^{32}	Marsaglia [8], also VAX
Los Alamos	5^{19}	0	2^{48}	2^{46}	Beyer, at workshop
Atari ST	3141592621	1	2^{32}	2^{32}	from the OS ROM
rand	1103515245	12345	2^{31}	2^{31}	Unix
drand48	25214903917	11	2^{48}	2^{48}	Unix and copies

It is by now well known that well-designed congruential generators have a k-dimensional output which fills a lattice in $[0, 1)^k$. All of these examples have adequate lattice structure in up to 8 dimensions [13, §2.4] and so satisfy the first two requirements, at least weakly. It hardly seems worth searching for "optimal" multipliers since most choices are reasonably good. (I find the criteria of Fishman and Moore [3] unreasonably stringent. Generators whose criteria differ by a factor of two are for practical purposes indistinguishable.) The worst performance here in terms of the ratio r of lattice sides is the Los Alamos generator with $r \approx 16.8$ in $[0, 1]^3$, but in absolute terms this is *much* better than all the shorter-period generators. It is not worth going beyond 8 dimensions, since 2^{32} points in $[0, 1]^8$ will be at least $\frac{1}{16}$ apart, and even 2^{48} will be $\frac{1}{64}$ apart.

Some of these generators (notably *Marsg*) have a poor two-dimensional discrepancy. This arises because one axis of the lattice of points from this generator is closely aligned with the axes in $[0, 1]^2$, the basis vectors being $2^{-32}(1, 69069)$ and $2^{-32}(-62184, 19400)$. The discrepancy is about that of independent 18-bit reals! The position of *LGM* is even worse; Fishman and Moore [3] quote its discrepancy as being in the interval $10^{-5}[1.488, 5.952]$, about that of independent 16-bit reals, whereas other generators with the same modulus and period can achieve 10^{-8}. I believe that discrepancy is not a good measure of a *pseudo*random number generator, and may not be a good measure of a *quasi*random sequence, since Koksma's inequality is only an upper bound for the accuracy of a quasi Monte Carlo integral.

The difficulty in implementing (1) is to perform the modular arithmetic rapidly. It is usually possible to implement *LGM* and *Marsg* in double-precision reals, and with suitable hardware this need not be outrageously slow (see Table 2).

```
REAL FUNCTION UNIF(SEED)
DOUBLE PRECISION SEED
SEED = MOD(69069.0D0*SEED+1.0D0, 4294967296.0D0)
UNIF = SEED*2.3283064D-10
END
```

For this to be valid the double precision needs at least 49 bits of precision, which even the most frugal implementations provide. Schrage [16] gave a portable integer implementation of *LGM*, improved in [2, p.202]:

```
REAL FUNCTION UNIF(ISEED)
INTEGER*4 ISEED,K
K = ISEED/127773/
ISEED = 16807*(ISEED-127773*K)-2836*K
IF (ISEED .LT. 0) ISEED = ISEED+2147483647
UNIF = 4.6566128E-10*ISEED
END
```

With floating-point hardware the double-precision method is often as fast.

In an attempt to construct a very portable pseudorandom number generator, Wichmann and Hill [18,19] combined three short-period congruential generators by addition modulo 1. On a machine with 32-bit integers their code is

```
REAL FUNCTION RND()
INTEGER X,Y,Z
DATA X,Y,Z/...,...,.../
```

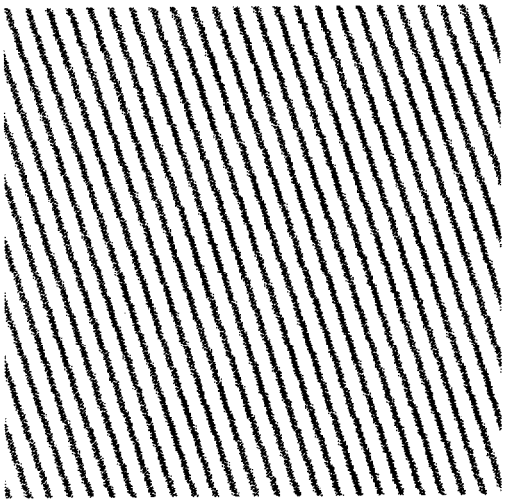

Fig. 1. Plot of all successive pairs (U_{2i}, U_{2i+1}) from one IBM PC BASIC generator.

```
X = MOD(171*X, 30269)
Y = MOD(172*Y, 30307)
Z = MOD(170*Z, 30323)
RND = MOD(X/30269.0 + Y/30307.0 + Z/30323.0, 1.0)
END
```

and the papers contain code for machines with 16-bit integers. This generator has period lcm(30268, 30306, 30322) = 6.96×10^{12}, but the three divisions and the mod make it rather slow on all the computers I use (Table 2). Thus I dispute that it is "efficient" (the title of [18]).

Some examples of congruential generators in common use have *both* too small a period and an incorrect implementation. Sawitzki [15] quotes one version of the built-in generator in IBM PC BASIC which was chosen as $a = 214013$, $c = 13523655$, $M = 2^{24}$ and hence would have a rather short period. What was actually implemented was

$$X_i = \left(214013 \left(X_{i-1} \bmod 2^{16}\right) + 13523655\right) \bmod 2^{24},$$

which has period 2^{16} but with numbers not uniformly spread in $\{0, \ldots, 2^{24} - 1\}$ (although the unevenness of the top 8 bits is small). The main problem is its performance in higher dimensions; for instance Fig. 1 shows a plot of successive pairs! Thus this generator will fail in many problems small enough to be programmed in BASIC.

One common objection to congruential generators with modulus $M = 2^\beta$ is that their "lower order bits have a small period". This applies to $\{X_i\}$ not $\{U_i\}$, and is only relevant if the generator is used to produce many bits per U_i. This seems a futile exercise, and usually takes longer than `INT(2.0*RND())` for each new bit!

Shift-register generators

The alternative family is that of *shift-register* generators. These generate L-bit integers by a recurrence of the form

$$Y_i = Y_{i-p} \oplus Y_{i-(p-q)}, \tag{2}$$

where $\oplus$ denotes bitwise exclusive or (the C operation ^, and available in most Fortran dialects but few Pascals). Recurrence (2) can be implemented by a simple circular buffer:

```
REAL FUNCTION RND()
PARAMETER(IP='p', IQ='q', IPQ=IP-IQ, L='L')
COMMON /RNDBLK/ Y(IP)
DATA I,J/IPQ,IP/
IY = XOR(Y(I), Y(J))
Y(J) = IY
I = I-1
IF (I .EQ. 0) I = IP
J = J-1
IF (J .EQ. 0) J = IP
RND = IY*(2.0**-L)
END
```

(The `XOR` function is valid in Sun Fortran; other systems have other conventions given in the Appendix.) Before use `Y(1),...,Y(IP)` are initialized with $Y_{-1}, \ldots, Y_{-p}$. Each of the L bits of Y_i then follows part of the sequence

$$b_j = b_{j-p} \oplus b_{j-(p-q)}, \tag{3}$$

with $j = i - t_k$ for bit k. Often this general form is called a *GFSR* but some authors consider only the case $t_k = (k-1)\tau$. For suitable p and q the sequence $\{Y_i\}$ has period $2^p - 1$, and if in addition the starting values are chosen suitably, the sequence is k-distributed for $k \leqslant \lfloor p/L \rfloor$ [5; 13, §2.3]. In practical terms this means that k-tuples $(U_i, \ldots, U_{i+k-1})$ have an almost independent distribution. One popular choice is $p = 521$, $q = 32$, $L = 32$, for which 16-tuples are practically independent, and single-precision reals U_i will be to a very good approximation uniformly distributed. As Table 2 shows, this generator can be implemented to run nearly as fast as a 32-bit congruential generator, so it comes close to meeting our aims in Section 1.

The snag with GFSRs is their initialization. The procedure given in [5] is trial-and-error, given in a modified form in [9]. One very simple way to initialize our example GFSR is to use the generator

$$b_j = b_{j-p} \oplus b_{j-q} \tag{4}$$

(which is just (3) run backwards). The first p values are arbitrary; then (4) is used to find p further values. Take the jth bit of `Y(i)` as b_{i+16j}, corresponding to $t_j = 16j$. Then the sequence $\{Y_i\}$ is 16-distributed, and, as Table 2 shows, the initialization is acceptably rapid. FORTRAN code is given in the Appendix. To reduce the $p = 521$ bits needed to initialize to a reasonable quantity we use a congruential generator (in fact *LGM* in Schrage's form).

Fushimi [4] considers a variant on this idea, with a time-consuming initialization but the same theoretical properties. Table 2 shows that initialization cannot be performed routinely since it takes as long as 250 000 calls. Nevertheless, initialization is acceptable on mainframes and C code can be more rapid; on the Atari it took 13 seconds for the second and subsequent initializations.

Initialization for GFSRs is still a subject of debate, and other ideas may yet be proved better. However, 16-distributed suffices for just about any conceivable practical purpose.

GFSRs have had a bad press. Knuth [6, p.30] warns against them:

> "*Caution:* Several people have been trapped into believing that this random bit-generation technique can be used to generate random whole-word fractions $(.X_0, X_1 \ldots X_{k-1})_2$, $(.X_k, X_{k+1} \ldots X_{2k-1})_2, \ldots$; but this is actually a poor source of random fractions, even though the bits are individually quite random."

In a similar vein, Marsaglia and Tsay [9] give several objections, including

> "The exclusive-or operation $\oplus$, is no faster than + or − in most computers so, taken with the poor statistical performance and relatively short periods, one wonders why $F(r, s, \oplus)$ generators have ever been given serious consideration.
> But they have. ..."

Here $F(r, s, \text{op})$ refers to the generator $Y_i = Y_{i-r}$ op Y_{i-s}. Many of the objections to GFSRs are based on extrapolation from inadequate examples, and some are of the nature of Marsaglia's original objection to congruential generators on the basis of their lattice behaviour (to abandon any method with a known potential flaw). I do not believe any of these objections stand serious scrutiny.

Shuffling

One often advocated way to improve a generator is to use shuffling. The best-regarded method is that of Bays and Durham [1] given in the following code:

```
      FUNCTION RAN0(ISEED)
      REAL V(98),Y
      LOGICAL FIRST
      DATA FIRST/.TRUE./
      IF (FIRST) THEN
        FIRST = .FALSE.
        DO 10 I=1,98
10        V(I) = RND(ISEED)
        Y = RND(ISEED)
      ENDIF
      J = 1+97*Y
      Y = V(J)
      V(J) = RND(ISEED)
      RAN0 = Y
      END
```

The idea is to keep an array of previously generated pseudorandom numbers, and to use the *previous* number to pick an element to return and replace. As Table 2 shows, the use of **RAN0** roughly doubles the timings. In Section 4 we will see an advantage of shuffling when running the same generator on different processors; if each is given a different seed they will be extremely unlikely ever to get into step.

3. Some timings

To illustrate why speed might be important, the examples of Section 2 were tested on a range of machines. Some attempt was made to code them as efficiently as possible in FORTRAN, although where the authors gave FORTRAN code this was used. The Atari ST is a 16-bit personal computer based on a 8MHz 68000; the Prospero FORTRAN compiler was used. As this has no floating-point hardware the times are dominated by the $U_i = X_i/M$ step which contributes around 120 μsecs. Probably all the generators except the FORTRAN version of *Marsg* and *WichHill* are acceptable.

The Amstrad 1640 is a 8086-based XT-class machine with an 8087 coprocessor. The IBM PS/2 model 60 had a 10MHz 80286 and 8MHz 80287 coprocessor. The times given were in similar proportion across a range of 8088/8087 to 80386/80387 machines. The Prospero compiler was used (so the extra mathematics functions of the 80387 are not used). Here a fast routine is desirable, and for serious use one should code a routine in assembler (but assembler for this chip family is notoriously unfriendly).

The Sun 3/160 has a 16.67MHz 68020 and 68881 and Weitek 1164/5 maths coprocessors. The *-fsoft* option uses the 68020 only, the *-f*68881 uses the 68881 and the *-ffpa* option uses the faster of the two coprocessors for each operation. With this hardware most generators are acceptable (but Wichmann–Hill is once again slow). As stated in Section 1, for some of my research, the fastest available routine is barely fast enough.

Timings in C are more difficult to report, partly due to the effects of its insistence on using double-precision arguments for functions, and partly from the inefficiency of the compilers used (Lattice C on the Atari ST, OS 3.2 on the Sun). (See Table 3.)

Table 2
Times for various generators on a range of computers; all times are in μsecs except initialization which is in seconds

	Atari ST	Amstrad	PS/2 60	Sun 3/160		
		1640	+80287	*-fsoft*	*-f*68881	*-ffpa*
log(U)	1670	225	180	1760	57	20
built-in [a]	210	200	120	185	78	47
Marsg (assembler)	60	—	—	17	17	10
BFSchrage	440	700	290	86	38	27
Marsg (FORTRAN)	2000	620	390	290	43	28
WichHill	1490	1220	550	370	84	46
GFSR521 init	1.25 s	3.50 s	1.25 s	0.12 s	0.12 s	0.10 s
per call	210	175	90	67	27	15
Fushimi [4] init	110 s	220 s	60 s	5.4 s	5.0 s	4.9 s
per call	360	345	145	79	36	23
Fushimi [b] init	50 s	150 s	37 s	5.0 s	4.1 s	4.0 s
per call	210	230	105	67	28	14
Shuffling [c]	250	190	—	87	34	14

[a] Varies from compiler to compiler.
[b] After some optimization of the code.
[c] The *additional* time for shuffling.

Table 3
Times for various generators in C; all times are in μsecs except initialization which is in seconds

	Atari ST	Sun 3/160	Comments
		-ffpa	
log(U)	3600	28	
rand (integer)	54	7.0	Returns $2^{31}-1$
rand (real)	300	11.5	Multiplied by $1/(2^{31}-1)$
drand48	1700	450	
random	—	20	Unix additive feedback generator
ran0	1100	40	Press et al. [12]
Fushimi init	24 s	3.7 s	
per call	330	22	

The Unix generator *rand* has been replaced by *drand48* (which is far too slow) and by a feedback generator *random* of the type $F(r, s, +)$. The shuffling generator *ran0* is an implementation of the Bays–Durham method applied to *rand*.

4. Parallel processors

Modern supercomputer architectures pose a particular problem, since they often require large numbers of random numbers generators to run (functionally) independently on separate processors. For example, the AMT DAP (formerly ICL DAP; [17]) has an array of 32^2 or 64^2 processors, and Transputer-based machines have networks of up to a few hundred processors. Vector machines such as CRAY X-MPs have a vector of processors, often 64.

An obvious idea is to distribute one generator across P processors, either by *decimation* (allocating processor p the sequence $U_i^{(p)} = U_{iP+p}$) or by using widely separated seeds (so $U_i^{(p)} = U_{i+t_p}$, $\{t_1, \ldots, t_P\}$ "uniformly spread" in $\{1, \ldots, P\}$). In either case note that we will need the independence of k-tuples with indices which are widely separated. With congruential generators it is both easy to generate decimated sequences ($\{U_i^{(p)}\}$ has multiplier a^P mod M and constant $(a^P-1)c/(a-1)$ mod M) and to test the lattice structure of any k-tuple $(U_{i+t_1}, \ldots, U_{i+t_k})$, so the theoretical properties of either scheme can be assessed.

It is well known that decimation is also very easy for GFSRs when P is a power of 2, since (by induction on ν) for $P = 2^\nu$ we have

$$Y_{iP} = Y_{(i-p)P} \oplus Y_{(i-(p-q))P},$$

so *if we decimate the initialization*, the same algorithm generates $\{U_i^{(p)}\}$. Care is needed to choose $(Y_{-pP}, \ldots, Y_{-1})$ suitably to achieve k-distribution of the important k-tuples.

The worry with these methods is the unwanted synchronization of generators on different processors if the number of uniforms per task is random. One way to avoid this is to increase the size of the statespace of each generator either by a shuffle or by a fixed (but different for each processor) permutation of each $\{U_i^{(p)}\}$.

The merits of these solutions will depend on the machine used. Congruential methods need considerable computation but little storage at each processor; GFSRs are light on floating-point computation but need over 2Kbytes of storage per processor.

Taking the base generator with period a power of 2 can cause problems. One study reported problems with a generator of period 2^{46} applied to a process on a 128^3 lattice sampled every 8192 steps and with decimation over 64 processors in a pipeline. The effective period in this problem is only 2^{12} steps and 2^{17} at each site, far too short. For a general-purpose generator on a supercomputer, a prime period is advantageous.

5. Conclusions

The reader may already have guessed which generators I actually use. For all but the most demanding problems *Marsg* suffices. It is simple to program in assembler (and I have done so on 6502, 68000 and 68020 chips as well as on a VAX) and easy to check against a double-precision implementation. If any doubt arises from the results of the simulation experiment I use GFSR521 as a cross-check, but have *never* had cause to doubt *Marsg*. However, the period of *Marsg* is beginning to seem too small, and GFSRs have the edge for the future, since they provide the fastest way to achieve an effectively infinite period *and* have demonstrably good theoretical properties.

The whole history of pseudorandom number generation is riddled with myths and extrapolations from inadequate examples. A healthy dose of scepticism is needed in reading the literature.

Acknowledgement

I am grateful to Masanori Fushimi for the C code for his generator and for correspondence on its implementation.

Appendix

The following FORTRAN code will initialize and run a shift-register generator with period $2^{521} - 1$ which is 16-distributed. It uses one nonstandard construction, the `XOR` function of Sun FORTRAN. On some machines this is called `IEOR` and on others

```
I = XOR(J,K)
```

is written as

```
I = J .XOR. K  or  I = J .NEQV. K
```

```
      function rndgf()
      integer p, q
      parameter(p = 521)
      integer src, dst, w(p)
      common /gfblk/ src, dst, w
      irnd = w(dst)
      rndgf = irnd * 4.6566128e-10
      w(dst) = xor(irnd, w(src))
```

```
      src = src-1
      if (src .eq. 0) src = p
      dst = dst-1
      if (dst .eq. 0) dst = p
      end

      subroutine gfinit(iseed)
      integer p, p2, q
      parameter (p=521, p2=p*2, q=32)
      integer src, dst, w(p), bit(31), x0(p2), wi
      common /gfblk/ src, dst, w
      print *, 'initializing'
      do 5 i = 1, p
5       x0(i) = 0
      bit(1) = 1
      do 10 j = 2, 31
10      bit(j) = 2*bit(j-1)
      im = bit(31)
      ix = iseed
      do 20 i = 1, p
        k = ix/127773
        ix = 16807*(ix - 127773*k) - 2836*k
        if (ix .lt. 0) ix = ix + 2147483647
        if (ix .gt. im) x0(i)=1
20      continue
      do 25 i = p+1, p2
25      x0(i) = xor(x0(i-p), x0(i-q))
      do 40 i = 1, p
        wi = 0
        do 30 j = 1, 31
30        wi = wi + bit(j)*x0(i+16*j)
40      w(i) = wi
      dst = p-q
      src = p
      print *,'done'
      end
```

References

[1] C. Bays and S.D. Durham, Improving a poor random number generator, *ACM Trans. Math. Software* **2** (1976) 59–64.

[2] P. Bratley, B.L. Fox and L.E. Schrage, *A Guide to Simulation* (Springer, New York, 1983).

[3] G.S. Fishman and L.R. Moore, An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31}-1$, *SIAM J. Sci. Statist. Comput.* **7** (1986) 24–45.

[4] M. Fushimi, Random number generation with the recursion $X_t = X_{t-3p} \oplus X_{t-3q}$, *J. Comput. Appl. Math.* **31** (1) (1990) 105–118 (this issue).

[5] M. Fushimi and S. Tezuka, The k-distribution of the generalized feedback shift register pseudorandom numbers, *Comm. ACM* **26** (1983) 516–523.

[6] D.E. Knuth, *The Art of Computer Programming, Vol 2, Seminumerical Algorithms* (Addison-Wesley, Reading, MA, 2nd ed., 1981).
[7] P.A.W. Lewis, A.S. Goodman and J.M. Miller, A pseudo-random number generator for the System/360, *IBM Sys. J.* **8** (1969) 136–145.
[8] G. Marsaglia, The structure of linear congruential sequences, in: S.K. Zaremba, Ed., *Applications of Number Theory to Numerical Analysis* (Academic Press, London, 1972) 249–285.
[9] G. Marsaglia and L.-H. Tsay, Matrices and the structure of random sequences, *Linear Algebra Appl.* **67** (1985) 147–156.
[10] S.K. Park and K.W. Miller, Random number generators: good ones are hard to find, *Comm. ACM* **31** (1988) 1192–1201.
[11] W.H. Payne, J.H. Rabung and T.P. Bogyo, Coding the Lehmer pseudorandom number generator, *Comm. ACM* **12** (1969) 85–86.
[12] W.H. Press, B.P. Flannery, S.A. Teubolsky and W.T. Vetterling, *Numerical Recipes in C* (Cambridge Univ. Press, Cambridge, 1988).
[13] B.D. Ripley, *Stochastic Simulation* (Wiley, New York, 1987).
[14] B.D. Ripley and M.D. Kirkland, Iterative simulation methods, *J. Comput. Appl. Math.* **31** (1) (1990) 165–172 (this issue).
[15] G. Sawitzki, Another random number generator which should be avoided, *Statist. Software Newsl.* **11** (1985) 81–82.
[16] L. Schrage, A more portable Fortran random number generator, *ACM Trans. Math. Software* **5** (1979) 132–138.
[17] J.D. Sylwestrowicz, Parallel processing in statistics, in: *Compstat 1982 Proc. I* (Physica-Verlag, Berlin, 1982) 131–136.
[18] B.A. Wichmann and I.D. Hill, Algorithm AS183. An efficient and portable pseudorandom number generator, *Appl. Statist.* **31** (1982) 188–190; correction: **33** (1984) 123.
[19] B.A. Wichmann and I.D. Hill, Building a random-number generator, *Byte* **12** (3) (1987) 127–128.